UNIX Code Migration Guide

UNIX Application Migration Guide

patterns & practices
proven practices for predictable results

Chapter 2: UNIX and Windows Compared

Larry Twork, Larry Mead, Bill Howison, JD Hicks, Lew Brodnax, Jim McMicking, Raju Sakthivel, David Holder, Jon Collins, Bill Loeffler
Microsoft Corporation

October 2002

Applies to:
Microsoft® Windows®
UNIX applications

Summary: Chapter 2: UNIX and Windows Compared provides the technical background behind the evolution of UNIX and Windows, pointing out the differences and similarities between the two platforms. It also looks at background technologies that can form part of your migration or coexistence plans. (38 printed pages)

Contents

Introduction

The *UNIX Application Migration Guide* was developed to help you migrate UNIX applications to the Microsoft® Windows® operating system. This guide provides the information you need to plan and budget your migration. It also examines methods for carrying out the migration and provides guidelines on implementing and managing the final product.

By migrating your application to the Windows platform, you benefit from a widely-used, full-featured operating system that runs the most popular business applications. With the addition of native UNIX packages such as Microsoft Interix, Windows can run many UNIX applications with a minimum of migration effort. The combination of the Windows platform and the Interix subsystem provides customers with a single enterprise platform on which they can run all of their Windows, UNIX-based, and Internet applications.

This guide is appropriate for managers, architects, and developers involved in the process of migrating an application from UNIX to Windows. Individual chapters cover the different aspects of the migration—including analysis, planning, porting of code, and testing of the migrated application.

This chapter gives an overview of the development and production environments in both Windows and UNIX. However, before you begin to plan your migration, it is important that you have a good understanding of both operating systems, their terminology, and the key differences between them.

Windows Evolution and Architecture

In the late 1980s, Microsoft began to design a new operating system that could take advantage of advances in processor design and software development. The new operating system was called Microsoft Windows NT® (for *new technology*). The current Windows 2000 and Microsoft Windows XP operating systems are based on Windows NT.

Figure 1 illustrates the evolutionary development of the Windows family of operating systems, culminating in today's Windows XP and soon in Microsoft Windows Server 2003. Windows XP is built on the robust and high-performance Windows NT kernel and incorporates many of the best features of Microsoft Windows 98 and Microsoft Windows Millennium Edition (Windows Me). These features include Plug and Play support, an intuitive user interface, and many innovative support services.

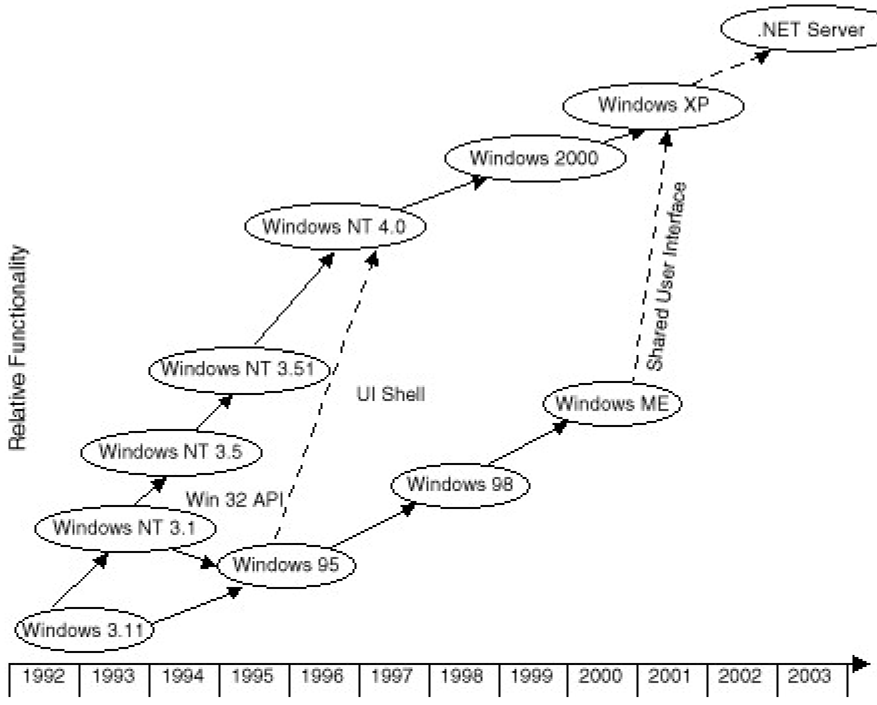

Figure 1. The evolution of the Windows family of operating systems

Windows NT Architecture

Windows NT architecture uses two processor access modes: *user mode* and *kernel mode*.

User mode includes application processes (typically Microsoft Win32® programs) and a set of protected subsystems. These subsystems are referred to as *protected* because each one is a separate process with its own protected virtual address space. The most important subsystem is the Win32 subsystem, which supplies much of the Win32 functionality to 32-bit Windows applications. The Windows subsystems, including the Win32 subsystem, are discussed in greater detail later in this chapter.

Another important subsystem, particularly with respect to UNIX applications, is the POSIX subsystem. POSIX stands for *Portable Operating System Interface* for computing environments, and consists of a set of international standards for implementing UNIX-like interfaces. POSIX began as an effort by the IEEE community to promote the portability of applications across different versions of UNIX. However, POSIX is not limited to the UNIX environment and has been implemented on a number of non-UNIX operating systems, including Windows NT. The POSIX subsystem implements these standards-based interfaces, and allows applications developers to more easily port their applications to Windows from another operating system.

Kernel mode is a highly privileged mode of operation where program code has direct access to all memory, including the address spaces of all user mode processes and applications, and to hardware. Kernel mode is also known as *supervisor mode*, *protected mode* or *Ring 0*. The kernel mode of Windows NT contains the *executive* as well as the system *kernel*. The executive exports generic services that protected subsystems call to obtain basic operating system services, such as file operations, input/output (I/O), and synchronization services. Partitioning of the protected subsystems and the executive simplifies the base operating system design and makes it possible to extend the features of an individual protected subsystem without affecting the kernel. The kernel controls how the operating system uses the processors, and performs operations such as scheduling, multiprocessor synchronization, and providing objects that the executive can use or export to applications.

Figure 2 is a high-level illustration of Windows NT architecture.

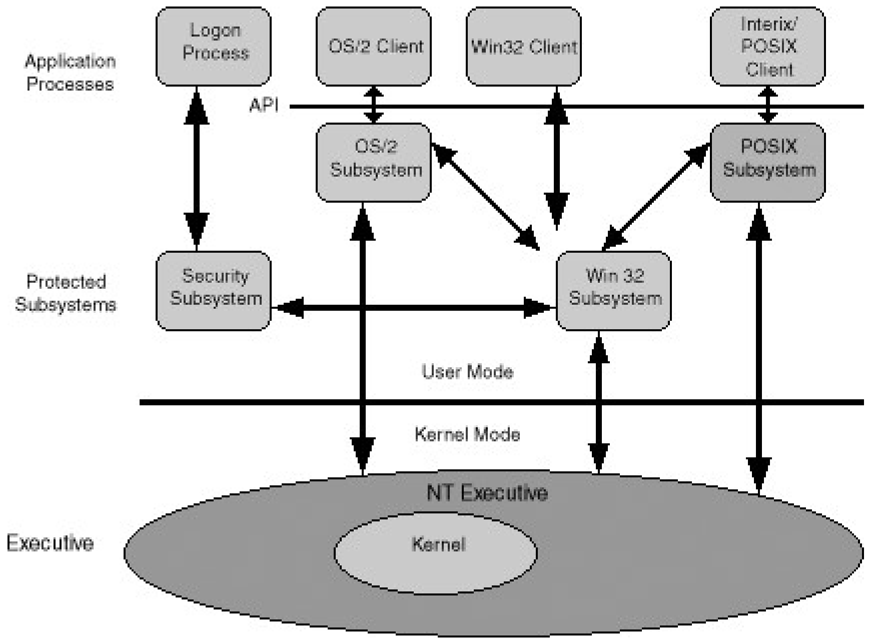

Figure 2. Windows NT architecture

The Windows operating system supports the following features and capabilities:

- Multitasking
- Choice of programming interfaces (subsystem and kernel application programming interfaces [APIs])
- Emphasis on graphical user interface (GUI) for users and administrators (the default user interface is graphical)
- Optional command-line interface
- Built-in networking (Transmission Control Protocol/Internet Protocol [TCP/IP] is standard)
- System services are provided by Windows Services
- Single compatible implementation

UNIX Evolution and Architecture

In 1969, Bell Laboratories developed UNIX as a *timesharing* system (the term used at that time to describe a multitasking operating system that supported many users at terminals). Although the first implementation was written in assembly language, the designers always intended for UNIX to be written in a higher-level language. Thus, Bell Labs invented the C language so that they could rewrite UNIX. UNIX has evolved into a popular operating system that runs on computers ranging in size from personal computers to mainframes.

Figure 3 shows the evolution of UNIX from a single code base into the wide variety of UNIX systems available today. In fact, this is only a summary–there are more than fifty flavors of UNIX in use today. The codes on the diagram refer to the brands and versions of UNIX that are in common use, including:

- AIX from IBM
- Solaris from SUN Microsystems
- HP-UX and Tru64 from Hewlett Packard
- UnixWare from Caldera
- Linux and FreeBSD, which are open source products

For further information, please refer to Chapter 4.

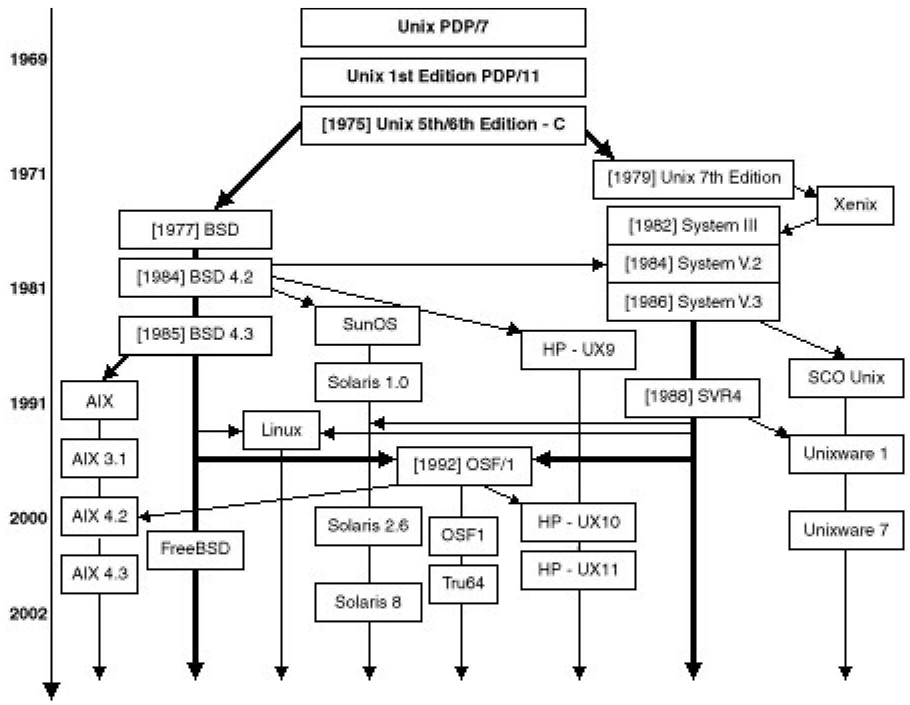

Figure 3. The history of UNIX implementations

UNIX became popular with computer manufacturers because it was written in a high-level language and was thus portable. Computer manufacturers could buy the rights to the UNIX source and modify it to make it run on their hardware. Although this portability greatly aided the acceptance of UNIX, it also created incompatible versions, which became one of the greatest problems for the developers of UNIX applications.

The architecture of UNIX can be divided into three levels of functionality, as shown in Figure 4. The lowest level is the *kernel* which schedules tasks, manages resources, and controls security. The next level is the *shell,* which acts as the user interface, interpreting user commands and starting applications. The highest level is *utilities,* which provides useful utility functions. (For more information about the shell, see the "Shells and Scripting" section later in this chapter.)

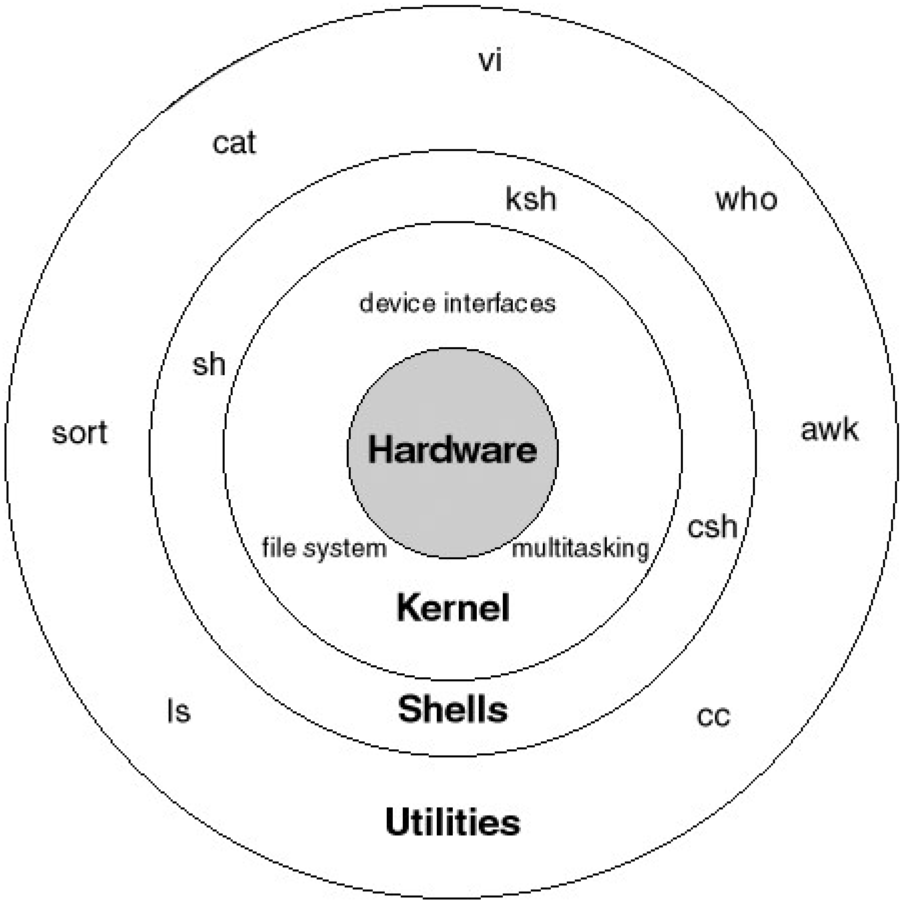

Figure 4. Structure of the UNIX operating system

The UNIX operating system supports the following features and capabilities:

- Multitasking
- Multiuser
- Kernel written in high-level language
- Programming interface
- Use of files as handles to reference devices and other objects
- Large number of simple tools
- Use of pipes and filters to undertake complex tasks through simple tools
- Default user interface is character-based
- Built-in networking (TCP/IP is standard)
- System services are provided through daemon processes
- Wide number of vendor platform implementations

The following sections explain these features in more detail.

Comparison of Windows and UNIX Architectures

This section compares and contrasts the Windows and UNIX architectures, emphasizing those areas that directly affect software development.

The areas covered in this section are:

- Kernels and APIs
- Hardware drivers
- Processes and threads
- Virtual memory management
- File systems and networked file systems
- Security
- Networking
- User interfaces
- System configuration
- Interprocess communication (IPC)
- DLLs and shared libraries
- Component-based development
- .NET
- Middleware
- Shells and scripting
- Development environments

Kernels and APIs

As do most operating systems, Windows and UNIX both have kernels. The kernel is responsible for all the basic functions of the operating system, such as:

- Creating files
- Starting processes
- Managing input and output
- Managing memory

In UNIX, the API functions are called *system calls*. System calls are a programming interface common to all implementations of UNIX. In most implementations, the functions defined by the system calls are the same; in some implementations, however, there are minor differences. Standards such as POSIX include a definition of the implementation of system calls, in addition to other features.

Similarly, Windows has an API for programming calls to the executive. In addition to this API, each subsystem provides a higher-level API. This approach allows the Windows operating systems to provide different APIs, some of which mimic the APIs provided by the kernels of other operating systems. The standard subsystem APIs include the Win32 API (the Windows native API) and the POSIX API (the standards-based UNIX API).

Objects and handles

As a Windows developer using the Win32 API, you use kernel objects to manage and manipulate resources such as files, synchronization objects, processes, threads, and pipes. Kernel objects are data structures maintained by the operating system kernel. To interact with a kernel object (and its associated resource), you must obtain a handle to the kernel object by calling the appropriate Win32 API. Regardless of the underlying resource type, the procedure for manipulating kernel objects is as follows:

1. Obtain a kernel object handle.

   For example, call the **CreateFile** function to open a file and obtain a file kernel object handle.

2. Manipulate the resource by using the kernel object handle.

   For example, call the **ReadFile** and **WriteFile** functions, supplying the handle as a parameter.

3. Close the handle when your work is complete.

   Call the **CloseHandle** function, irrespective of the handle type.

Windows subsystems

A subsystem is a portion of the Windows operating system that provides some service to application programs through a callable API. The subsystems run in separate processes and do not share virtual memory. Therefore, a subsystem must send messages to another subsystem to communicate with it. All messages pass through the executive, which performs a security check to guarantee that the subsystems do not interfere with one another.

Subsystems come in two varieties, depending on where the request is finally handled:

- *Environment subsystems* execute in user mode and provide functions through a published API. The best known environment subsystem is Win32, which provides an API for operating system services, GUI capabilities, and functions to control all user input and output.
- *Integral subsystems* perform key operating system functions and execute as part of the executive or kernel. The best known of the integral subsystems are the security subsystem and the virtual memory manager. Other subsystems include the object manager, the process manager, and the I/O manager.

The Win32 subsystem

The Win32 subsystem allows applications to benefit from the full power of the Windows family of operating systems. Win32 has a vast collection of functions, including the capabilities required for advanced operating systems, such as security, synchronization, virtual memory management, and threads. By using the Win32 API, you can write applications that run on all versions of Windows while taking advantage of capabilities that exist only on later versions.

The Win32 API is grouped into six categories:

- **Base services**

  Base services are functions that let applications use the features of the operating system, such as memory management, file systems, devices, processes, and threads. An application uses these functions to manage and monitor the resources that it needs to complete its work. For example, an application uses memory management functions to allocate and free memory. Process management and synchronization functions start and coordinate the operation of multiple applications or multiple threads within a single application.

- **Common control library**

  A common control library implements a set of common controls shown as windows. Applications use these controls to maintain consistency with the Windows shell and to maintain the distinctive Windows behavior and appearance. Common controls range from fairly simple, such as combo box and status bar controls, to complex, such as calendar and tree view controls.

- **Graphics Device Interface**

  The Graphics Device Interface (GDI) provides functions and data structures that applications use to generate graphical output for displays, printers, and other devices. GDI enables applications to draw geometric shapes, such as lines, curves, and closed figures and to manipulate text and images. GDI allows the application to control visible attributes, such as color and style, when drawing shapes and text. Applications can direct output to a physical device or to a logical device such as memory or a metafile.

- **Network services**

  Network services provide functions for network management and Windows networking (WNet). Network management lets a systems administrator or network manager create and manage shared resources, such as directories, network printers, and users. Windows networking functions enable applications to query and control network connections and to retrieve information about the current network configuration. These functions are independent of any network provider or physical network implementation.

- **User interface**

  User interface functions give applications the means to create and manage a user interface. Applications use these functions to create and use windows to display output, prompt for user input, and interact with the user. The behavior and appearance of windows that an application creates are controlled by window classes and corresponding window procedures. A window class defines default characteristics, such as whether the window processes mouse button clicks or has a menu. The corresponding window procedure contains code that defines the behavior of the window in response to events and user input.

- **Windows shell**

  Windows shell functions enable applications to use the shell interfaces and to enhance various aspects of the Windows shell. A *context menu* handler is a shell extension that modifies the contents of a shortcut menu. The system displays a shortcut menu when the user clicks an object with the right mouse button. The shortcut menu contains commands that apply specifically to the object that was clicked. Most shortcut menus contain a **properties** command that displays the property sheet for the selected object. A *property sheet* contains information about the object in a set of overlapping or tabbed windows called *pages*. A property sheet handler is a shell extension that adds pages to the system-defined property sheet. The system uses icons to represent files. The default icon displayed is the same for all files with the same extension. An *icon handler* can override the default and display a different icon for some files.

> **Note** The APIs provided by different environment subsystems cannot be mixed. A file opened in the POSIX subsystem is not compatible with the API in the Win32 subsystem. For this reason, you must use special techniques when linking different subsystems.

The POSIX subsystem and Interix

Windows NT, Windows 2000, and Windows XP provide a fully standards-compliant subsystem that supports programs written for the POSIX portable operating system environment. Programs written for the POSIX environment on any other operating systems should perform in exactly the same manner on Windows. Although the POSIX subsystem is standards compliant and provides the majority of the system calls found in UNIX implementations, not all UNIX applications are POSIX compliant.

To add more comprehensive support for UNIX programs, Windows provides the Interix subsystem. Interix is a multiuser UNIX environment for a Windows-based computer. Interix conforms to the POSIX.1 and POSIX.2 standards. It provides all of the features of a traditional UNIX operating system, including pipes, hard links, symbolic links, UNIX networking, and UNIX graphical support through the X Window System (also called X Windows). It also includes case-sensitive file names, job control tools, compilation tools, and more than 300 UNIX commands and utilities, such as KornShell, C Shell, awk, and vi. See Chapter 10 for further information about Interix features and commands.

Because the Interix subsystem is layered on top of the Windows kernel, it is not an emulation; it is a native environment subsystem that integrates with the Windows kernel, just as the Win32 subsystem does. When you install Interix, you install a new extended subsystem that replaces the POSIX subsystem provided with Windows and that provides true UNIX functionality. Shell scripts and other scripted applications that use UNIX and POSIX.2 utilities run under Interix. (For more information about shell scripts, see the "Shells and Scripting" section later in this chapter.)

These behaviors of the Interix environment are different from open systems:

- Interix has no superuser.
- Interix has different user authentication.

  User and group information is stored in the Windows Security Access database. While the database stores both users and groups, group names and user names must be unique; that is, no group can have a user's name and vice versa. (This database replaces the /etc/passwd and /etc/groups files or Network Information Service [NIS] map files in UNIX.) Users can belong to many groups.

- Interix supports user name mapping.

  Interix uses user name mapping to associate Windows users with user identifiers (UIDs) and group identifiers (GIDs). Mapping allows the actual user and group names to appear as the file owner and file group when a long directory listing is requested.

Hardware Drivers

The Windows Driver Model provides a platform for developing drivers for industry-standard hardware devices attached to a Windows-based system. The keys to developing a good driver package are to provide good setup and installation procedures and to provide interactive GUI tools for configuring devices after installation. In addition, hardware must be compatible with Windows Plug and Play technology to ensure a user-friendly hardware installation. If hardware manufacturers meet these and other requirements, they can display the "Designed for Windows" logo on their packaging and documentation.

In some versions of Windows, the user must reboot the computer after installing new hardware, drivers, and peripherals. Windows XP, however, has features that eliminate the need to reboot if drivers are signed with a digital certificate. This certificate indicates that a driver has passed the Windows Hardware Compatibility Tests, which ensure that the driver functions correctly with the Windows operating system.

In UNIX, there are several different ways to manage drivers. Some UNIX implementations allow for dynamic loading and unloading of drivers, whereas other implementations do not. The UNIX vendor usually provides drivers. On Intel platforms, the range of supported hardware for UNIX is typically smaller than that for Windows.

Process Management

Multitasking operating systems—such as Windows and UNIX—must manage and control many processes at once. Each process has its own code, data, system resources, and state. Resources include virtual address space, files, and synchronization objects. Threads are a part of a process; each process has one or more threads running on its behalf. Like a process, a thread has resources and a state associated with it. The Windows and UNIX operating systems both provide process and threads.

The following sections provide more detail on how UNIX and Windows manage processes.

Multitasking

UNIX was designed to be a multiprocessing, multiuser system. At any point in time, a user may have many processes running on UNIX. Consequently, UNIX is very efficient at creating processes.

Windows has evolved from its beginnings on Microsoft MS-DOS®, which did not support preemptive multitasking. As a result, Windows relies heavily on threads instead of processes. (A thread is a construct that enables parallel processing within a single process.) Creating a new process in Windows is a relatively expensive operation.

Multiple users

One key difference between UNIX and Windows is the implementation of multiple users on one computer.

On UNIX, when a user logs on, a shell process is started to service the user's commands. The UNIX operating system keeps track of users and their processes and prevents processes from interfering with one another. Because all the processes run on the server, the resource demands on the computer can grow quite large, especially with many users and large applications.

On Windows, when a user logs on interactively, the Win32 subsystem's Graphical Identification and Authentication dynamic-link library (GINA) creates the initial process for that user, known as the user *desktop*. This desktop is where all user interaction or activity takes place. Only a particular instance of the logged-on user has access to the desktop. This allows the user to control the computing environment (sometimes known as the *shell*). Other users are not intended to be able to log on to that computer at the same time. However, if a user uses Terminal Services or Citrix, Windows can operate in a server-centric mode similar to UNIX. (For more information about Terminal Services and Citrix, see the "Windows Terminal Services and Citrix" section later in this chapter.)

Multithreading

Most new UNIX kernels are multithreaded to take advantage of symmetric multiprocessing (SMP) computers. Initially, UNIX did not expose threads to programmers. However, POSIX does have user-programmable threads. In fact, POSIX has two different implementations of threads, depending on the POSIX version.

In Windows, creating a new thread is very efficient. Windows applications are able to use threads to take advantage of SMP computers and to maintain interactive capabilities when some threads take a long time to execute.

Fibers

Windows has another unit of execution, called *fibers*, which UNIX does not have. Fibers are sometimes referred to as lightweight threads. Fibers must be manually scheduled by a thread, and they run in the context of that thread. Fibers are usually used in applications that service a large number of users, such as database systems. Fibers do not provide much improvement in speed over threaded applications, but they do provide a good technique for porting applications that are designed to schedule their own threads.

Process hierarchy

When a UNIX application creates a new process, the new process becomes a child of the creating process. This process hierarchy is often important, and there are system calls for manipulating child processes.

Unlike UNIX, Windows processes do not share a hierarchical relationship. The creating process receives the process handle and ID of the process it created so a hierarchical relationship can be maintained/simulated if the application requires it to do so. However, the operating system treats all processes as belonging to the same generation.

> **Note** Both Windows and UNIX processes (by default) inherit the security settings of the creating process.

Signals, exceptions, and events

UNIX and Windows have mechanisms by which processes can indicate an event or error. In both operating systems, these events are signaled by a form of software interrupts. In UNIX, these mechanisms are called *signals* and are used for normal events, simple interprocess communication, and abnormal conditions such as floating point exceptions. Windows has two separate mechanisms, as follows:

- An *events* mechanism handles expected events, such as communications between two processes.
- An *exception* mechanism handles non-standard events, such as the termination of a process by the user. Computer hardware may generate exceptions such as invalid memory access and math errors. Windows uses a facility named Structured Exception Handling (SEH) to handle these exceptions.

Filters and pipes

UNIX introduced a philosophy of computing that incorporates features known as filters and pipes. A well-designed UNIX program gets its input from the standard input stream and writes its results to standard output. This makes the program a *filter*, analogous to a water filter or a filter in engineering. The filter has one input and one output and performs an operation on information passing through it. *Pipes* give users the ability to link these filter programs together so that the output of one program is fed into the input of another. A typical use of this capability is sorting; that is, running one program that generates some desired output and piping the output into the sort utility for viewing.

Daemons and services

In UNIX, a daemon is a process that the system starts to provide a service to other applications. Typically, the daemon does not interact with users. UNIX daemons are started at boot time from init or rc scripts.

A Windows service is the equivalent of a UNIX daemon; it is a process that provides one or more facilities to client processes. Typically, a service is a long-running Windows application that does not interact with users and consequently does not include a user interface. Services may start when the system boots and they continue running across logon sessions. Services are controlled by the Service Control Manager (SCM), and one of the few requirements for writing a service is that it must communicate with the SCM to handle starting, stopping, and installing.

Because it runs in a separate process, a service runs in user mode with a specific user identity. The security context of that user determines the capabilities of the service. Most services run as the Local System account. This account has elevated access rights on the local computer but has no privileges on the network domain. If a service needs to access network resources, it must run as a domain user with enough privileges to perform the required tasks. On UNIX, a daemon runs with an appropriate user name for the service that it provides or as the special user named nobody.

Summary of processes and threads

Table 1 summarizes the differences between Windows and UNIX in terms of processes and threads.

Table 1. Windows and UNIX processes and threads

Feature	Windows	UNIX
Primary mechanism	Threads	Processes
Processes	Yes	Yes
Threads	Yes	Yes, but different implementations
Fibers	Yes	No
Performance	Very good at creating threads	Very good at creating processes
Process hierarchy	No	Yes
Security inherited	Yes	Yes (except setuid)

Virtual Memory Management

Both UNIX and Windows use virtual memory to extend the memory available to an application beyond the actual physical memory installed on the computer. In UNIX, virtual memory is handled by the kernel; in Windows, virtual memory is handled by an executive service. Virtual memory uses a number of techniques to:

- Inform the application that additional memory is available.

- Transparently enhance system performance (and therefore application performance) by reading for disk as efficiently as possible.

Virtual memory uses areas on disk to extend real memory. In addition, the virtual memory manager moves program and data files from the hard disk into physical memory only when the files are needed. Because virtual memory is managed by the operating system and is transparent to applications, there should be no need to consider virtual memory during the migration process.

File Systems and Networked File Systems

This section describes the file system characteristics of UNIX and Windows. Table 2 shows the basic features of modern file systems.

Table 2. File system characteristics

Feature	Description
File names	User-defined name associated with the physical file, typically 255 characters or more
Directories	Named folders to store files in, usually arranged in a hierarchical, tree-like structure
Path names	Way of referring to a specific file or directory in a particular place
Aliases, Links, Shortcuts	Methods for pointing one file at another or giving a file multiple names
Security	Method of protecting and controlling access to files and directories
File information	Method of storing the properties of a file, such as creation date, modification time, size, and location on disk

Both UNIX and Windows support many different types of file system implementations. Some UNIX implementations support Windows file system types, and there are products that give Windows support for some UNIX file system types.

File names and path names

Everything in the file system is either a file or a directory. UNIX and Windows file systems are both hierarchical, and both operating systems support long file names with up to 255 characters. Almost any character is valid in a file name, except the following:

- / in UNIX
- ?, ", /, \, >, <, *, |, and : in Windows

In UNIX, there is a single directory known as the *root* at the top of the hierarchy. You locate all files by specifying a path from the root. The UNIX notation for file paths is a series of directory names separated by a single slash mark, followed by the file or directory name. The root directory is named /, so a path begins with /; for example, /etc/passwd. Paths can also be specified as relative to the current working directory (which is represented as ".") or to the parent of the current directory (represented as "..").

UNIX makes no distinction between files on a local hard drive partition, CD-ROM, floppy disk or networked file system. All of the files appear in one tree under the same root. For this to work, UNIX uses a process called *mounting.* New file systems (for example, a hard drive partition) are mounted on an empty directory and then appear as a seamless part of the file system directory tree.

Windows can have many hierarchies; for example, one for each partition and one for each network drive. Like UNIX, a path in Windows is defined by a series of directories and a file name. Unlike UNIX, the separator is a backslash, and the drive name (for example, C or D) or UNC name (for example, \\SERVER\SHARE) may also need to be specified. However, Windows can use "." and ".." just as UNIX does.

UNIX file system features

UNIX treats all files as steams of data with no boundaries or structure. In UNIX, each file in the file system is described by an *inode.* This differs from a file name, as it refers to the following information about the file:

- Permissions
- Owner
- Type
- Date and time of creation and of last access and modification
- Size
- Pointers to the data blocks allocated to the file

The inode does not contain the name of the file. A directory contains the file names and associated inodes. UNIX can also create *hard links*, which allow a file to appear in more than one directory with more than one name.

In the UNIX file system, devices are also represented by files. Device files are usually found in the /dev directory. For example, you can run a program and ignore all of its output by redirecting the output to the null device, /dev/null. It is also possible to send data directly to a serial port or terminal by using this technique. Some versions of UNIX even expose memory and running processes in this manner (/dev/mem and /dev/proc, respectively).

Applications, not the operating system, handle file structures. This imparts a simplicity and uniformity to input/output but can be a performance issue for large files or busy systems if not handled carefully.

Networked file systems

File systems do not have to be stored on a local drive (for example, a hard disk or CD-ROM). Users and applications can access them over the network from a server or peer computer. To do this, the operating system uses special file systems—called *networked file systems*—that work over the network.

The Network File System

The standard UNIX network file system is the Network File System. Developed by Sun Microsystems, the technology is licensed to most of the UNIX vendors. NFS is designed to integrate into the UNIX file system model. An NFS server exports a directory, and an NFS client then mounts that exported directory just as it would a local file system. To the user, the networked file system appears to be just another part of the directory tree.

UNIX also has an automount mechanism. Automount directories are automatically made available when an application attempts to access them. They are then unmounted after a period of inactivity. The automount mechanism reduces the number of network file systems mounted and simplifies administration.

NFS is a client/server implementation. The actions that are executed on the server are minimal. The server does not keep any state associated with the client; all the state data is kept on the client. This method of retaining state ensures that the server can perform quickly and efficiently but places many requirements on the client.

Server message block and Common Internet File System

One of the earliest implementations of network resource sharing for the MS-DOS platform was network basic input output system (NetBIOS). Features in NetBIOS allowed it to accept disk I/O requests and direct them to file shares on other computers. The protocol used for this was named server message block (SMB). Additions were made to SMB to apply it to the Internet, and the protocol today is known as Common Internet File System (CIFS).

In Windows, the server shares a directory, and the client then connects to the UNC for that share. Each network drive usually appears with its own drive letter, such as X.

Windows and UNIX Network File System interoperability

UNIX and Windows can interoperate by using NFS on Windows or CIFS on UNIX. There are a number of commercial NFS products for Windows. For UNIX, in addition to commercial implementations of CIFS, a software option called Samba is widely used. Samba is an alternative to installing NFS client software on Windows-based computers for interoperability with UNIX-based computers. Samba is an open-source, freeware, server-side implementation of a UNIX CIFS server. To provide file and print services, it implements security in the form of authentication and authorization. It also implements NetBIOS-style name resolution and browsing.

Summary of file system differences

The preceding sections discussed the architectures of the UNIX and Windows file systems, which are both hierarchical but differ in many details. Table 3 summarizes the differences between the Windows, Windows with Interix, and UNIX file systems.

Table 3. Summary of file systems differences

Feature	Windows	Windows/Interix	UNIX
Overall structure	Hierarchal, multiple trees	Hierarchal, single tree	Hierarchal, single tree
Drive names	Yes (C, D)	Yes, under /dev/fs (for example, /dev/fs/C)	No
Mounting partitions	Yes	Yes	Yes
Path separator	\	/	/
Case-sensitive names	No	Yes	Yes
Hard links	No	Yes	Yes
Symbolic links	No	Yes	Yes
Shortcuts	Yes	No	No

Network file system	SMB		NFS
Device files	No	Yes, with exceptions (for example, /dev/mem)	Yes
Set user ID	No	Yes	Yes
Security	ACLs	Mapping between bit permissions and ACLs	Simple bit permissions

Security

UNIX and Windows architectures differ in many ways, including security implementations. The following subsections describe some of these security implementation details and differences.

User authentication

A user can log on to a computer running UNIX by entering a valid user name and password. Some UNIX implementations require optional extra credentials, such as smart cards (for example, with pluggable authentication modules on Solaris and Linux). A UNIX user can be local to the computer or known on an NIS *domain* (a group of cooperating computers). In most cases, the NIS database contains little more than the user name, password, and group.

A user can log on to a computer running Windows by entering a valid user name and password. In addition, Windows can require optional credentials such as certificates and smart cards. A Windows user can be local to the computer, known on a Windows NT domain or known in the Active Directory® directory service. The Windows NT domain contains only a user name, password, and user groups. Active Directory contains the same information as the Windows NT domain, and may contain contact information for the user, organizational data, certificates and so on.

UNIX security

UNIX uses a simple security model. The operating system applies security by assigning permissions to files. This model works because UNIX uses files to represent devices, memory, and even processes. Security permissions are applied to users or to groups.

In most cases, users are people who log on to the system, but users can be special users such as system services (daemons). In UNIX, each user has a UID, which (unlike in Windows) does not have to be unique. A user is logged on to the system when a shell process is running that has that user's UID. Groups are sets of users. A UNIX group has a GID. Every process has a UID and a GID associated with it.

> **Note** The credentials that a user supplies when logging on is usually a user name and a password. Some implementations of UNIX support the use of smart cards for interactive logon. Smart cards support cryptography and secure storage of private keys and certificates, enabling the strong authentication of users.

Security permissions

When a user logs on to the system by entering a user name and a password, UNIX starts a shell with the UID and GID of that user. From then on, all access to files and other resources is controlled by the permissions assigned to the UID and GID or the process. The UIDs and GIDs are configured in two files, /etc/passwd and /etc/group.

Each file in the file system has a bitmap that defines its permissions. The permissions grantable are read, write, and execute. These permissions are grouped in three sets: the owner of the file, the owner's group, and everybody else (world). A full (long) listing for a file shows the file permissions as a group of nine characters that indicate the permissions for owner, group, and world. The characters **r**, **w**, **x**, and **-** are used to indicate read, write, execute, and no permission, respectively. For example, if the owner of a file has all permissions but the group and world have only read permission, the string is as follows:

```
rwxr--r--
```

> **Note** Some UNIX implementations have extended the basic security model to include access control lists (ACLs) similar to those used in Windows. However, ACLs are not implemented consistently across all versions of UNIX.

Effective UID and effective GID

There are occasions when a process started by a particular user must access resources that the user does not have permissions to access. UNIX has a mechanism to handle this situation. Processes can have *effective* UIDs and GIDs that are different from the UID, the GID, and the parent process. An effective UID or GID is one that the operating system uses for the duration of the process.

Network Information System

The UNIX operating system was originally designed to run on a server by itself and not on a network, in a manner similar to stand-alone Windows-based computers. When computers can access resources on other computers on a network, synchronization of users (UIDs) and groups (GIDs) across computers becomes a problem. If the numerical identifiers are not properly synchronized, access requests across the network could incorrectly identify the user or group, which would result in a security breach.

The Network Information System (NIS) solves this problem by using a client/server model for processing requests. One computer on a domain is designated the *master computer*. Computers that serve as backups to the master are known as *subordinate computers*. All other computers on the domain are clients. When a client application must check credentials, the call is forwarded to the master computer, instead of being processed locally as it would on a computer not running NIS. The master looks up the user information in a database file called a *map* and returns the results.

Windows security

Windows uses a unified security model that protects all objects from unauthorized access. The system maintains security information for:

- **Users**. The people who log on to the system, either interactively by entering a set of credentials (typically user name and password) or remotely through the network. Every user's security context is represented by a logon session. Each process that the user starts is associated with the user's logon session.
- **Objects**. The secured resources that a user can access. For example, files, synchronization objects, and named pipes represent kernel objects.

Figure 5 illustrates the Windows security model and the relationship between the process-level access token, the object's security descriptor, and the DACL for the security descriptor.

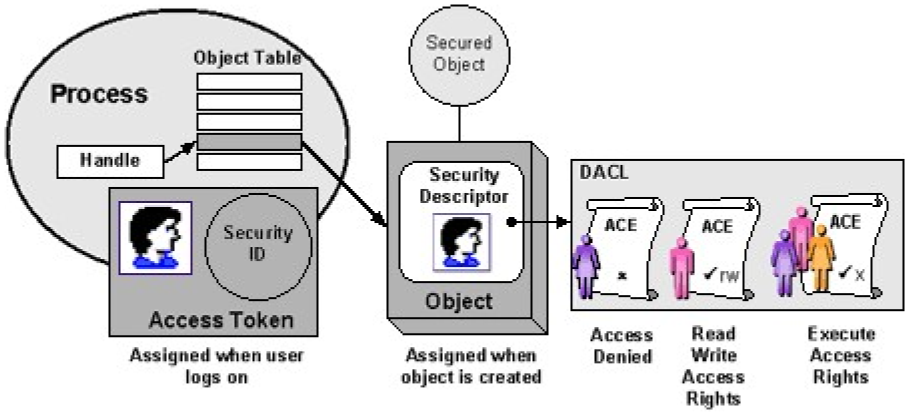

Figure 5. The Windows security model

Access tokens

An access token is a data structure associated with every process that is started by a particular user (and is associated with that user's logon session). The access token identifies who the user is and what security groups he or she is a member of. Although users and groups have human-readable names to ease administration, they are uniquely identified internally (for performance reasons) by security identifiers (SIDs).

Security descriptors

A security descriptor describes the security attributes of each object. The information in the security descriptor includes the owner of the object and a discretionary access control list (DACL). The DACL contains a list of access control entries (ACEs) that define the access rights for particular users or groups of users. The owner of the object controls the DACL and uses it to determine who should and should not be allowed access to the object, and what rights they should be granted.

The security descriptor also includes a system access control list (SACL), which is controlled by system administrators. Administrators use SACLs to specify auditing requirements for object access. For example, an administrator can establish a SACL that specifies the generation of an audit log entry whenever a user attempts to delete a particular file.

The sequence of events from the time a user logs on, to the time he or she attempts to access a secure object, is as follows:

1. The user logs on by entering a set of credentials. The system validates these credentials by comparing them against information maintained in a security database (or Active Directory).
2. If the user is authenticated, the system creates a logon session that represents the security context for the user. Every process created on behalf of the user (starting with the Windows shell process) contains an access token that describes the user's security context.
3. Every process subsequently started by the user is passed a copy of the access token. If one process results in additional processes, all child processes obtain a copy of the access token and are associated with the user's single logon session.
4. When a process (acting on behalf of the user) attempts to open a secure object such as a file, the process must initially obtain a handle to the object. For example, when attempting to open a file, the process calls the **CreateFile** function. The process specifies a set of access rights on the call to **CreateFile**.

5. The security system accesses the object's security descriptor and uses the list of ACEs contained in the DACL to find a group or user SID that matches one contained in the process's access token. When this task is complete, the user is either denied access to the object (if a deny ACE is located) or the user is granted a specific set of access rights to the object. The granted rights may be the same as the rights initially requested or may be a subset of the rights initially requested. For example, the **CreateFile** call can request read and write access to a file, but the DACL may allow only read access.

Impersonation

When a thread within a process attempts to access a secured object, the security context that represents the user making the access attempt is normally obtained from the process-level access token.

You can, however, associate a temporary access token with a specific thread. For example, within a server process, you can impersonate a client's security context. The act of impersonation associates a temporary access token with the current thread. The temporary impersonation token represents the client's security context. As a result, the server thread uses the client's security context when it attempts to access any secured object. When the temporary access token is removed from the thread, impersonation ceases and subsequent resource access reverts to using the process-level access token.

Active Directory

Windows 2000 introduced Active Directory, a directory service that is used to store information about objects. The objects can include users, computers, printers, and every domain on one or more wide area networks. Active Directory can scale from a single computer to many large computer networks. Active Directory provides the store for all domain security policy and account information. It replaces the flat account namespace in earlier versions of Windows with a hierarchical namespace for user, group, and computer account information.

Windows 2000 also introduced new authentication protocols based on Internet standards, including Kerberos Version 5 and Transport Layer Security (TLS). For backward compatibility, Windows 2000 supports existing NTLM authentication protocols.

Windows implementation of secure channel security protocols (Secure Sockets Layer [SSL] 3.0/TLS) supports strong client authentication by mapping user credentials in the form of public-key certificates to existing Windows NT accounts. Administrators use common administration tools to manage account information and access control, whether the administrators are using password authentication or certificates. External users who do not have Windows 2000 accounts can be authenticated through public-key certificates and mapped to an existing Windows account. This allows businesses to give trading partners limited or full access to their internal network.

Networking

The primary networking protocol for UNIX and Windows is TCP/IP. The standard programming API for TCP/IP is called *sockets*. Sockets were created for UNIX at the University of California, Berkeley. Sockets provide an easy-to-use, bidirectional stream between systems across a network. The Windows implementation of sockets is formally known as Windows Sockets but is usually called Winsock. Winsock conforms well to the Berkeley implementation, even at the API level. Most of the functions are the same, but slight differences in parameter lists and return values do exist.

User Interfaces

The UNIX user interface was originally based on a character-oriented command line, whereas the Windows user interface was originally based on a GUI. This difference is a result of the background of the two operating systems. UNIX originated at a time when graphic terminals were not available; Windows was (as the name suggests) designed to take advantage of advances in the graphics capabilities of computers. However, both UNIX and Windows now support a mixture of character and graphical interfaces.

The UNIX character-based interface

The standard UNIX shells and tools are all character based and command line oriented. For the UNIX shells and UNIX-based applications to be able to communicate with different models of character terminals, they must be aware of the different functions available and the command sets for each terminal.

Termcap and terminfo

To minimize the amount of specific terminal information embedded in a program, UNIX has databases of terminal capabilities; these databases are known as *termcap* and *terminfo*. Rather than embedding terminal commands into an application, developers can use program libraries provided with the operating system to query the database for specific movement commands, thus allowing their applications to operate with a variety of hardware.

Curses

Another application development package specifically designed to alleviate the problem of terminal dependence is the *curses* library originally written at the University of California, Berkeley. It is a set of functions for manipulating terminal input and output (mostly output). These perform such actions as clearing the screen, moving the cursor to a specific row and column, and writing a character or string to the screen. There are also input functions to retrieve user input in various modes, such as read one character and read a string terminated by carriage return. Curses and similar libraries enable developers to create highly interactive, character-based applications, such as text editors.

X Windows and Motif

The standard windowing system for UNIX systems is the X Window System (or X Windows) developed at MIT. X Windows is a platform-independent, basic windowing system. It consists of a lower level API called X library (or Xlib) and a higher-level library called X Toolkit Intrinsics. X Windows separates the server (which manages the display of graphical information) from the client (which is the application program that uses X Windows). The server and client can run on separate computers, so the application may run on a powerful numerical server while the output appears on a graphics workstation. This feature has also led to the development of *X terminals*—that is, computers equipped only to display graphics on a computer screen.

Because X Windows is a set of toolkits and libraries, it does not have graphical user interface standards as do Windows and Mac OS. Motif is the most common windowing system, library, and user interface style built on X Windows. Motif handles windows and a set of user interface controls known as *widgets*. Widgets cover the whole range of user interface, including buttons, scroll bars, menus, and high-functionality items such as a Web browser widget.

Windows Terminal Services and Citrix

Windows can provide sessions that run applications on a server but are displayed on a client workstation. These sessions can be implemented with both Terminal Services (on Windows 2000) and Citrix.

Both Terminal Services and Citrix use a server-based session similar to the way UNIX operates. The difference is that Terminal Services and Citrix use a smart GUI terminal specific to running Windows programs. This is analogous to the way an X terminal operates in a UNIX environment. System managers can use Terminal Services to deliver Windows functionality to a low-end computer or even one that does not run Windows. Terminal Services can also be used to remotely administer a Windows-based server.

Terminal Services is particularly useful to implement server-based applications in a thin client environment. Additionally, Terminal Services provides a smart GUI protocol that works effectively on slow links. This protocol allows enterprises to consolidate applications in a remote location, without the loss of performance usually associated with slower remote networks.

System managers can implement Terminal Services using network load balancing in scale-out server clusters. This configuration allows for both higher availability and the ability to add more servers when the load increases.

Applications that use Terminal Services or Citrix usually fall into two categories:

- Desktop applications (such as Microsoft Office) moved from the desktop client to a central server
- Remote applications that require thin client connectivity and that are unable to operate through a Web-based interface

System Configuration

UNIX users generally perform system configuration by editing the configuration files with any of the available text editors. Many UNIX users and system administrators like the fact that much of the configuration for UNIX is stored in text files. The advantage this is that the user does not need to learn a large set of configuration tools; he or she must only be familiar with an editor and possibly a scripting language. The disadvantage is that the information in the files comes in various formats, so the user must learn the various formats to change the settings.

To manage a network, UNIX system administrators often employ scripts to reduce the possibility of repetition and error. In addition, administrators can use NIS to centralize the management of many standard configuration files. Although different versions of UNIX have GUI management tools, such tools are usually specific to each version of UNIX.

Windows has GUI tools for configuring the system. The advantage of these tools is that they can offer capabilities depending on what is being configured. In recent years, the Microsoft Management Console (MMC) has provided a common tool and user interface for creating configuration tools. Windows also provides a scripting interface for most configuration needs through the Windows Scripting Host (WSH). WSH implements two widely known scripting languages—Microsoft Visual Basic® Scripting Edition (VBScript) and Microsoft JScript®—plus a set of objects for manipulating system configuration settings. (WSH is described in greater detail later in this chapter.)

Startup scripts and logon/logoff scripts

In UNIX, scripts are used at startup time to invoke most system and user processes. Such scripts include any special scripts a systems manager has written, in addition to all the system services (such as networking and printing). UNIX has a special process called **init** that the kernel starts. The init process is responsible for starting all other services and process. It is configured through a file called /etc/inittab. For BSD-style systems, init runs various rc scripts to configure services, and for System V–style systems, init runs scripts under the /etc/rc?.d directory. Configuration of the characteristics of any service is carried out within /etc/inittab and the rc scripts.

In Windows, the startup characteristics of different services (such as network servers and print servers) are controlled through a GUI and stored in the registry. There is no need to create a script to start or stop services. In Windows NT and Windows 2000, logon scripts can run each time a user logs on. The script can be used to configure the environment for the user, for example to provide access to network shares and printers. A logon script is usually a batch file or a WSH script, and can be shared among several users. Logon scripts can be assigned through the User Manager, or a user policy can be set to run a script for all users through the Policy Editor. A user policy can also be used to set a logoff script.

Interprocess Communication

An operating system designed for multitasking or multiprocessing must provide mechanisms for communicating and sharing data between applications. These mechanisms are called interprocess communication (IPC). Some forms of IPC are designed for communication among processes running on the same computer, whereas other forms are for communicating across the network between different computers.

UNIX interprocess communication

UNIX has several IPC mechanisms that have different characteristics and are appropriate for different situations. Shared memory, pipes, and message queues are all suitable for processes running on a single computer. Shared memory and message queues are suitable for communicating among unrelated processes. Pipes are the mechanism usually chosen for communicating with a child process through standard input and output. (For more information about message queues, refer to the "Message Queues" section later in this chapter.)

For communication across the network, sockets are usually the chosen technique. Migration from UNIX sockets to Windows sockets is s usually a straightforward process involving few changes to the code.

Windows interprocess communication

Windows has many IPC mechanisms, some of which have no counterpart in UNIX. As with UNIX, Windows has shared memory, pipes, and events (equivalent to signals). These are appropriate for processes local to a computer. The shared memory implementation is based on file mapping, because certain forms of shared memory can be used across the network. Named pipes can also be used for network communications.

Other IPC mechanisms supported by Windows are the clipboard/Dynamic Data Exchange (DDE), Component Object Model (COM), and send message. These are mostly used for local communications, but DDE and COM both have network capabilities. Windows sockets and Message Queuing (also known as MSMQ) are good choices for cross-network tasks.

Two additional IPC mechanisms for Windows are remote procedure call (RPC) and mailslots. RPC is designed for use by client/server applications and is most appropriate for C and C++ programs. Mailslots are memory-based files that a program can access by using standard file functions. Mailslots have a fairly small maximum size. Usage is often similar to named pipes except that mailslots are effective for broadcasting small messages.

Synchronization

Both UNIX and Windows have an extensive set of process and thread synchronization techniques. Both operating systems use *semaphores*, which are synchronization primitives used to control access to a resource that can support a limited number of users. Both UNIX and Windows also use mutex objects to control mutually exclusive access to a resource.

For lightweight control of multithread access to a section of code, Windows offers critical section objects. Critical sections are similar to mutexes, but access is limited to the threads of a single process. This makes them appropriate for controlling access to a shared resource. Threads can access the critical section in any order, but the order is not guaranteed.

Message queues

In UNIX, a message queue is an IPC mechanism. One application sends messages to the queue; another application reads messages from the queue. The queues are memory based and are very fast as a result. However, the messages will disappear if the system fails. Message queues were introduced in AT&T System V UNIX. Because of this, many versions of UNIX that are based on BSD may not have them. POSIX has message queues but the API is not exactly the same as in System V.

Windows provides a reliable messaging system called Message Queuing (MSMQ). Message Queuing provides guaranteed message delivery, efficient routing, security, and priority-based messaging. In essence, a Message Queuing message is guaranteed to be delivered, but there is no specific guarantee about exactly when it will be received. The operation is the same as on UNIX—one application writes to the queue and another reads from it. The API, however, is completely different.

Shared memory

As mentioned previously, both Windows and UNIX provide shared memory as one of the IPC mechanisms. Both mechanisms are intended to provide a section of memory that can be shared between processes to pass data and control information; however, the implementation details are different.

In one of the UNIX implementations, the program must first call a function to get a shared memory identifier, *shm_id,* for the amount of shared memory. The identifier is then used in calls to attach the shared memory to the process. There are other functions for controlling and removing the shared memory. This type of shared memory mechanism was introduced in the AT&T System V.2 version of UNIX.

Later UNIX versions introduced shared memory based on the concept of file mapping. The **mmap** function sets up a segment of memory that can be read or written by two or more programs. This mechanism is used to manipulate files. The **mmap** function creates a pointer to a region of memory associated with the contents of the file that is accessed through an open file descriptor.

Windows implementation of shared memory is based entirely on the concept of file mapping. A common section of memory can be mapped into the address space of multiple processes. If no file is specified in the creation function, the shared memory is allocated from a section of the page file. As in the UNIX implementation, which uses an identifier, Windows uses a *handle* identifier to identify the memory that is mapped into the process at the requested address.

Both the UNIX and Windows file mapping solutions offer the capability of saving the contents in a permanent file.

Pipes

Pipes have similar functionality on both Windows and UNIX systems. Their primary use is to communicate between related processes.

UNIX pipes can be named or unnamed. They also have separate read and write file descriptors, which are created through a single function call. With unnamed pipes, a parent process that must communicate with a child process creates a pipe that the child process will inherit and use. Two unrelated processes can use named pipes to communicate.

Windows pipes can also be named or unnamed. A parent process and a child process typically use unnamed pipes to communicate. The processes must create two unnamed pipes for bidirectional communication. Two unrelated processes can use named pipes, even across the network on different computers. Typically, a server process creates the pipe, and clients connect to the bidirectional pipe to communicate with the server process.

DLLs and Shared Libraries

Windows and UNIX both have a facility that allows the application developer to put common functionality in a separate code module. UNIX calls this feature a *shared library*. Windows calls this feature a *dynamic-link library (DLL)*. Both allow application developers to link together object files from different compilations and to specify which symbols will be exported from the library for use by external programs. The result is the ability to reuse code across applications. The Windows operating system and most Windows programs use many DLLs.

Component-based Development

The Windows platform offers developers a wide range of component-based development tools and technologies, which are discussed in the following sections.

Component Object Model

COM is Microsoft's first component-based development technology. Developers can use COM to develop component-based software by exploiting a set of well-defined development techniques and runtime services. By adhering to the COM development model and by using one of the many COM-aware development environments, developers can easily build component-based software that is capable of interacting with other components developed by different organizations, potentially in different development languages.

Although many of the required development techniques—such as how functionality should be exposed through interfaces—are complex, the development environments available on the Windows platform mask this complexity. One of the most popular development environments is Visual Basic.

Some of the key features of the COM programming model are as follows:

- COM objects expose functionality through well-defined interfaces, the binary format of which is defined by the COM specification. (This functionality matches the classic C++ virtual function table [v-table] layout in memory.)
- An interface consists of a set of methods (although most development environments also allow properties to be exposed at the interface level through a pair of **property-get** and **property-set** methods).
- COM supports component versioning.
- COM components can be hosted in process (through DLLs), out of process (through executable files) or in executable files on remote computers.
- All COM components and COM interfaces on a particular computer are logged centrally in the Windows registry, a hierarchical configuration database for the Windows platform.

The Windows registry contains information such as what hardware is on the system, how the hardware and system are configured, and what applications are installed on the system. The registry replaces the myriad of .ini files prevalent on earlier versions of Windows. It has better performance than these files, provides a convenient central location to store all this data, and provides fine-grained security. Each registry key can be protected with an ACL in exactly the same way that files can be protected.

For COM, the registry stores a globally unique identifier (GUID) to identify each component class and interface installed. GUIDs are 128-bit integers that are guaranteed to be unique. COM uses this information to determine which component class to create when an application requests that an object (component) be instantiated.

Each component also has a user-friendly name known as a ProgID or programmatic identifier, that is created by the component vendor and that is not guaranteed to be unique. The recommended format for a ProgID is *vendor*. *component*. *version*, where *vendor* and *component* are alphanumeric names.

When an application must use an object, it starts by calling a COM function, **CoCreateInstance**, to create the component. This function takes the registered GUID for the object class (CLSID) as an argument. If the developer chooses to use the user-friendly ProgID instead, it first calls a function to get the CLSID from the ProgID. The application may also pass the initial interface GUID to **CoCreateInstance**, or it may pass a null entry to receive the default interface. COM finds the server for the class, loads the class into memory if necessary, and marshals the call if the server is in another process or across the network.

After a COM component is created, it can be queried for a particular interface that the application needs to perform its work. Because the interfaces are identified by GUIDs just like the components, the **QueryInterface** call takes the GUID as an argument and either returns the interface requested or returns a null entry if the interface is not implemented by the class.

For more information about COM, see the Microsoft COM Technologies home page.

COM+

COM+ (formerly Microsoft Transaction Server [MTS]) is based on COM and adds a series of infrastructure-type services designed to help you build sophisticated, component-based distributed systems. Most of the COM+ services do not require many—if any—additional lines of code in your components. Instead, COM+ introduces declarative attributes, which you can use to inform the COM+ executive of the services that your component requires at run time. Some of the key COM+ services are:

- **Distributed transaction processing**

  This service is used by components that update databases or by other resource managers, such as Message Queuing. The COM+ distributed transaction service ensures that all actions associated with a given transaction complete successfully, or the entire transaction fails. This all-or-nothing model of work management ensures the consistency of an application's state, even across multiple distributed databases.

- **Resource management and pooling**

  As applications start to scale to larger numbers of clients, objects in the application must share critical (and limited) resources, such as network connections, database connections, threads, and memory. COM+ provides a number of resource-management and pooling features to improve scalability. These features include thread pooling, object pooling, and database connection pooling.

- **Queued Components**

  The Queued Components service provides an asynchronous message-based communications model—an essential requirement for distributed systems. Whereas a conventional COM method call is a synchronous operation, a call to a queued component results in an asynchronous message being dispatched. The message is reliably delivered by the underlying services of Message Queuing. One of the advantages of this service is that it removes the need for the server (component) and client to run simultaneously. For example, if the server that hosts the target component is currently offline or unreachable through the network, the message request is queued and is subsequently passed to the component when the server comes online.

- **Publish and subscribe event delivery**

  The COM+ Loosely Coupled Event (LCE) service allows applications to publish information to subscriber applications and components. The LCE service provides a level of indirection between information publishers and information subscribers. Publishers communicate directly with the LCE service (rather than directly with subscribers), whereas subscribers register their interest in particular information types by notifying the LCE service. This approach means that publishers do not need to be concerned with the identity of subscribers and vice versa.

- **Role-based security**

  You can use the COM+ role-based security to perform authorization within your component by checking role membership. For example, you may need to restrict certain functionality within a component to specific groups of users, such as managers. You can use COM+ to define application-level roles (such as "managers"), populate them with user accounts at deployment time, and then either programmatically or declaratively (through attributes) check role membership to enforce authorization decisions.

- **Concurrency management**

  COM+ provides an automated concurrency management system that relieves you from the complex task of writing the synchronization logic required to handle concurrent client requests in a multiuser environment.

For more information about COM+, see the Microsoft COM + home page.

.NET components

Microsoft .NET is Microsoft's latest component-based development platform. Although from a high-level perspective it facilitates component-based development in a similar fashion to COM, it radically extends the development platform and provides the tools and technologies that developers can use to develop a new kind of Internet-based distributed application.

.NET is based on open Internet standards, which include:

- Hypertext Transfer Protocol (HTTP) for conveying message-based requests and responses across the Internet.
- Extensible Markup Language (XML) for defining data. XML is self-describing, structured data in text form. XML can represent any structured data that in the past has been in a different form, such as datasets from database queries. With XML, an application can get data from a database or other data source, process it as necessary, and send it to another application across the network.
- Simple Object Access Protocol (SOAP) for remote object communication across the Internet. You can think of SOAP as an RPC mechanism for use on the Internet. Because the payload of a SOAP message is represented as XML and is passed over HTTP, messages can be passed through firewalls—a critical problem with conventional RPC mechanisms. Assuming that the receiving application correctly authenticates the sender, the receiving application can process the request and return a response as a separate SOAP message.

.NET also encompasses COM+ services (though they are referred to as Enterprise Services in .NET), which you can exploit through an efficient interoperability layer. You can use this same layer to continue to use existing COM components and Win32 DLLs from .NET applications. You can also call .NET components directly from Win32/COM-based code.

.NET provides a set of technologies that you can use to develop applications for many different device types, including a myriad of different hand-held devices, desktop computers, and large-scale server systems.

.NET services

.NET services provide information to applications in much the same way that Web sites provide information to users of Web browsers. .NET services create a framework for sharing information between applications and devices, typically by using SOAP as the underlying delivery mechanism. The ability to find .NET services is provided by well-known, global directory services, such as the emerging Universal Description, Discovery, and Integration (UDDI) directory service.

.NET services are platform independent because they are based on Internet standards. They are independent of programming language, application, platform, and operating system.

The .NET Framework

The Microsoft .NET Framework is the platform for building, deploying, and running Internet-based distributed applications. It introduces a new programming model that developers can use to build XML-based .NET services and applications.

The .NET Framework provides the necessary foundation, thus permitting developers to concentrate on solving business problems, writing business logic, and creating user interfaces. It also solves many traditional application deployment issues and facilitates the operation of Internet-scale and enterprise-scale applications.

The primary elements of the .NET Framework are the common language runtime and a base class library. The common language runtime provides a managed runtime execution environment for .NET Framework applications. It provides many features traditionally associated with operating systems. Some of the key features are:

- Loading and executing code
- Just-in-time compilation of Microsoft intermediate language (MSIL) to native code
- Application memory isolation and management
- Security
- Strong type-checking
- Access to type metadata
- Cross-language exception handling
- Interoperability with existing code in COM objects and Win32 DLLs
- Other developer support services that include debugging and runtime profiling

The .NET base class library provides an integrated set of classes that expose the underlying functionality of the Win32 API. All classes are language independent and can be used by all .NET languages, including the new Microsoft Visual C#™ .NET and Microsoft Visual Basic .NET, in addition to traditional C++.

You can use this flexibility to choose the language and tools best suited to the job or the ones with which you have the most experience. Different teams of developers on a project can choose different languages, but they can still share their code and create new subclasses from classes written in a different language. This code reuse can dramatically increase team productivity and decrease development costs.

Other core .NET technologies include:

- **Microsoft ADO.NET**

  You can use ADO.NET, a data access technology, to access a host of different data stores, including Microsoft SQL Server™, Active Directory, and many other OLE DB–aware or Open Database Connectivity (ODBC)–aware databases. ADO.NET extends traditional data access models and includes features designed to support the inherently disconnected nature of Web applications.

- **Microsoft ASP.NET**

  You can use ASP.NET to rapidly build traditional Web applications and also Web services.

- **Windows Forms**

  You can use Windows Forms (WinForm) classes to build traditional GUI-based Windows applications.

The .NET Framework is designed so that designers of both Web and rich client Windows Form (WinForm) applications have similar tools and features available to them. The goal is to provide a rapid application development environment to developers whether they are creating an ASP.NET Web application, a .NET service or a Windows Forms application. The Windows Forms designer and the ASP.NET page designer both feature drag-and-drop placement of controls and separation of code from visual presentation.

For more information about .NET, see the Microsoft .NET home page.

For more information about UDDI, see the UDDI.org Web site.

Middleware

The following subsections compare the various middleware solutions available for UNIX and Windows applications.

OLTP systems

Online transaction processing (OLTP) systems have been implemented in UNIX environments for many years. These systems perform functions such as resource management, threading, and distributed transaction management. OLTP systems typically provide support for multiple languages and development environments.

- Common OLTP systems include:
- BEA Systems' Tuxedo
- NCR Corporation's Top End
- Transarc's Encino for DCE (distributed computing environment)

Although OLTP was originally developed for UNIX, many OLTP systems have Windows versions. Additionally, gateways exist to integrate systems that use different transaction monitors—for example, the Tuxedo gateway to Top End.

The current challenges for OLTP systems relate to how to integrate with Web and e-business systems. Many OLTP systems have provided a bridge to the Java programming language, and provide gateways to Common Object Request Broker Architecture (CORBA) and COM.

When considering transaction and resource management during a UNIX migration, developers should remember that OLTP systems provide many of the same features as COM+. As with most cross-platform products, OLTP monitors achieve these features by introducing new APIs to the development environment. Introducing COM+ for transaction and resource management during a migration can lessen this type of dependency.

Queuing systems

As mentioned earlier in this chapter, message queuing is provided as a feature in AT&T System V UNIX, and can be achieved through sockets in Berkeley UNIX versions. These types of memory queues are most often used for interprocess communications and do not meet the requirements for persistent store and forward messaging.

To meet these requirements, versions of IBM's MQSeries and BEA Systems' MessageQ (formally

DEC's MessageQ) are available for UNIX. A reliable and resilient store and forward message queue provide a key building block for enterprise integration and highly available, loosely coupled systems.

Microsoft provides similar functionality with Message Queuing for Windows. IBM and BEA Systems also provide versions of their queuing systems for Windows. Gateway products exist that bridge the various queuing systems.

The reasons for a migration to Windows may include the need to integrate with commercial off-the-shelf applications. The queuing system for such a migration would need to provide an API that easily integrates into these applications. For example, Message Queuing provides for a COM Automation Interface API and .NET classes.

Enterprise Application Integration systems

The need for increased overall efficiency of application infrastructures has led to the need to integrate what were formally stand-alone applications. E-business systems have added requirements for integration outside the enterprise's firewall.

An approach to accomplish this integration has been to create an infrastructure that manages invoking the stand-alone applications and integrates the data transfer between these applications. Enterprise Application Integration (EAI) systems provide this type of solution.

EAI systems are typically cross-platform systems that provide bridge technology to OLTP monitors (such as Tuxedo), message queuing systems (such as MQSeries), and distributed object models (such as COM and CORBA). In this way, an EAI system integrates with the stand-alone application on the application's own terms, and then provides a data transfer mechanism between applications.

A weakness of EAI systems has traditionally been the need to include compiled interface definition language (IDL) to achieve the required data marshaling. Microsoft BizTalk® Server provides this type of functionality based on XML as the common language for information interchange. XML eliminates the need for compiled IDL for each application interface.

If a conversion from UNIX requires this type of loosely coupled system integration functionality, you should seriously consider using XML for data interchange in the migration architecture. Bridging IDL with XML may require you to create an adapter application. However, you can create an adapter application once for any particular system, rather than for each interface.

Shells and Scripting

A shell is a command-line interpreter that accepts typed commands from a user and executes the resulting request. In addition to executing programs, shells usually support advanced features, such as the ability to recall recent commands and a built-in scripting language for writing programs.

Programs written through the programming features of a shell are called *shell scripts*. In addition to scripts written through the use of shells, there are also languages specifically designed for writing scripts. As with shell scripts, these scripting languages are interpreted. The use of scripting languages leads to rapid development (often with relaxed syntax checking) but slower performance.

Windows and UNIX support a number of shells and scripting languages, some of which are common to both operating systems.

Command-line shells

On the Windows platform, **Cmd.exe** is the command prompt or the shell. With the command prompt, a user can run programs or scripts and invoke applications. The command prompt has a memory or buffer for recent commands, so the user can retrieve, run, and edit them using various techniques.

On UNIX, a number of standard shells provide the UNIX user interface. These shells include:

- **The Bourne shell (sh)**

  This is the simplest shell, often set as the default. It can invoke programs and create pipes, but it has no command memory or advanced scripting capabilities.

- **The C shell (csh)**

  This shell includes command memory and a scripting language similar to the C language. A Windows version of the C shell comes with the Interix product.

- **The Korn shell (ksh)**

  The Korn shell also features command memory and a built-in language for creating script files. The Korn shell is based on the Bourne shell but includes additional features, such as job control, command-line editing, functions, and aliases. Windows versions of the Korn shell are delivered with the Windows Services for UNIX (SFU) and Interix products.

Scripting languages

The following subsections explain the scripting languages and scripting language support provided in Windows and UNIX.

Windows Scripting Host

WSH is a language-independent environment for running scripts and is often used to automate administrative tasks and logon scripts. WSH provides objects and services for scripts, establishes security, and invokes the appropriate script engine depending on script language. Objects and services supplied allow the script to perform tasks such as displaying messages on the screen, creating objects, accessing network resources, and modifying environment variables and registry keys.

WSH natively supports VBScript and JScript. Other languages that are available for this environment are Perl, Rexx, and Python. WSH is built in to all versions of Windows after Microsoft Windows 95. It can also be downloaded or upgraded from Microsoft.

Perl

Perl is an acronym for Practical Extraction and Report Language. It is an interpreted language that was originally designed for UNIX, but has since been ported to many platforms. Perl provides a cross-platform scripting environment that developers can use to write scripts that can be run on both Windows and UNIX. Perl is effective for string manipulation. Although Perl is not delivered with Windows, there are many sources for versions of Perl that are designed to run on Windows. Perl does come with the SFU and Interix products.

Rexx

Rexx is an acronym for Restructured Extended Executor Language, and was originally developed by IBM UK Laboratories. It is a procedural language that is designed for application programs to use as a macro, or scripting, language. Although Rexx can issue commands to its host environment and call programs and functions written in other languages, it is designed to be independent of a specific operating system. There are versions available for UNIX and Windows.

Python

Python, like Perl, is an interpreted language. It has many similar features to Perl but has a clearer programming structure and syntax, making Python code easier to read and maintain. Although it was designed for UNIX, it is now widely available on other platforms, including Windows. Python is object oriented and includes dynamic data structures and typing. Python is ideal for rapid software development where maintainable code is important. Python is not shipped with Windows, but can be downloaded from the Python Web site.

Tcl/Tk

Tcl/Tk is yet another interpreted language. Like Perl, it is effective for string manipulation, and is available across UNIX and Windows platforms. Tcl/Tk is particularly applicable to the development of cross-platform GUIs. Tcl/Tk is not shipped with Windows, but can be downloaded from the Tcl/Tk Web site.

Development Environments

The development environments for UNIX and Windows have many similarities. In both UNIX and Windows, you have a choice of environments. The generic UNIX development environment uses a set of command-line tools. However, there are many third-party integrated development environments (IDEs) for UNIX, some of which are designed to be cross-platform environments. On Windows, you have two main choices of development environments: a native Windows development environment and a UNIX-like development environment such as Interix.

Because this guide is designed to help developers migrate UNIX applications to Windows, the following subsections focus only on the Windows development environments.

Standard Windows development environment

The standard Windows development environment uses the Microsoft Platform Software Development Kit and Microsoft Visual Studio®.

Platform Software Development Kit

The Platform SDK delivers documentation for developing Windows applications, libraries, headers, and definitions needed by language compilers, samples with code, and command-line and stand-alone tools for Windows and kernel development. The Windows SDK and the Microsoft .NET Enterprise Server SDK are combined to form the Platform SDK.

> **Note** The Platform SDK is available at no cost on the Microsoft MSDN Web site or as a CD.

The SDK documentation includes developer guides and references for all Windows APIs, including Win32, COM+, and GDI+, along with many others. There are also guides and references for the .NET Enterprise Server APIs, including BizTalk Server, Microsoft Exchange Server, and SQL Server. Development guides contain information on designing applications for all recent versions of Windows, including the 64-bit versions of the Windows Server 2003 family. Documentation and header files exist for the following categories of APIs and services:

- Base services
- Component services
- Data services
- Graphics and multimedia services
- Messaging and collaboration services
- Networking and directory services
- Security
- Setup and system administration
- Tools and scripting
- User interface services
- .NET services
- Windows API

The Platform SDK includes a rich set of command-line and stand-alone Windows tools for building, debugging, and testing applications. Tool categories delivered with the Platform SDK are:

- Cryptography
- Debugging
- DirectX
- File management
- MAPI (Messaging API)
- Multimedia
- OLE
- Performance
- Resource files
- TAPI (Telephony API)
- Testing

Visual Studio

Visual Studio is an IDE that delivers a complete set of tools for application development, including the development of multitier components, user interface design, database programming and design, and development team support. Visual Studio provides language tools, editing tools, debugging tools, performance analysis tools, and application installation tools.

Visual Studio has compilers and development tools for several popular languages, including C, C++, and Visual Basic. Microsoft Visual Studio version 6.0 also includes support for Java language applets, applications, and components through Microsoft Visual J++®. Microsoft Visual Studio .NET includes the new language, C#. Both versions come with database support and a real database for developing applications. Support for each language includes the IDE with editor and common toolbox, compiler, linker, and debugger. The common environment reduces costs associated with training and eliminates the disorienting effects of switching languages.

For Web applications, Visual Studio delivers tools for development by distributed teams. Visual Studio

6.0 includes Microsoft Visual InterDev™, an integrated tool for creating Web applications through Hypertext Markup Language (HTML), script, and components. The components can be developed in any of the available languages—C, C++, Visual Basic or Visual J++. Visual Studio .NET integrates Web design even more completely into the C# and Visual Basic environments. Expanding on the popular rapid application development (RAD) capabilities of Visual Basic, Visual Studio .NET lets developers use drag-and-drop tools to add Web components on a page. Capabilities for .NET services—including the editing features of Microsoft IntelliSense®—are fully integrated. Developers can easily create .NET services, deploy them, and use them in other applications, whether the applications are Web based or client based.

Visual Studio also provides performance analysis tools that enable developers and testers to understand the structure and flow of the application and to isolate performance bottlenecks. In Visual Studio 6.0 , the performance analysis tool is Visual Studio Analyzer; in Visual Studio .NET, the tool is Application Center Test.

In addition, Visual Studio includes package and deployment tools to enable developers to deploy components and functionality for distributed applications.

The Interix development environment

The Interix Software Development Kit contains documentation, tools, API libraries, and headers needed by language compilers for porting UNIX applications to Windows. With the Interix SDK, you can host your own tools and applications alongside SFU tools and applications.

Included in the Interix SDK is a UNIX development environment, with tools such as the GNU gcc, g++ and g77 compilers, and the gdb debugger. The Interix SDK also provides user interfaces, through the cc and c89 compiler drivers (that is, interfaces to the compiler and linker programs CL.exe and Link.exe, respectively) for Microsoft Visual C++ version 5 and later, with which you can compile C programs to provide the benefits of the native compiler for Windows. The cc and c89 utilities work only with the Visual C++ compiler; they do not work with gcc. You cannot compile C++ code by using the cc and c89 interfaces. You must use g++ for C++ code.

The SDK documentation includes developer guides and references for all POSIX.1 system interfaces and headers, Interix extensions to POSIX.1 and POSIX.2 interfaces, and the International Organization for Standardization/American National Standards Institute (ISO/ANSI) C libraries. Development guides contain information about designing and building UNIX daemons as services, curses, and X Windows–based applications, and porting UNIX code, as well as documentation and header files for the following categories of APIs and services:

- POSIX.1 APIs
- Cryptography
- User interface services
- Curses and terminal routines
- X Windows
- Database (**dbm**)
- RPCs
- Sockets
- Memory-mapped files
- System V IPC mechanisms
- BSD string and memory functions
- Pseudo terminals

- Controlling terminals
- Security
- Setup and System Administration
- Tools and Scripting

Interix provides a rich set of command-line and stand-alone tools for building, debugging, and testing applications. Tool categories delivered with the SDK are:

- Compiling (**cc**, **c89**, **gcc**, **g++**, and **g77**)
- Linking (**ld**)
- Debugging (**gdb**)
- File management
- Performance
- Testing

Interix integration with Visual C++

If Visual C++ is installed, SFU Setup will configure the Interix SDK to work with Visual C++.

If Visual C++ is installed after the Interix SDK, the location of the Visual C++ compiler and linker is provided manually to the cc(1) and c89(1) utilities. The developer does this by using the **Windows System Properties** dialog box to create a Windows system variable named INTERIX_COMPILERDIR and setting its value to the path of the directory where Visual C++ is installed, in POSIX format. For example, if Visual C++ is installed in directory C:\MSDEV, the value of INTERIX_COMPILERDIR would be /dev/fs/C/MSDEV. If the path contains spaces, the MS-DOS version of the path should be used.

Conclusion

Windows provides all the features that make it the right choice for organizations that want to run all their applications on a single desktop. On the Windows platform, line-of-business and office productivity applications can run side by side and exchange data seamlessly. Earlier UNIX applications can be ported to run under Windows, or they can use Interix or other migration environments to run the applications with minimum modification. In either case, users do not need to switch environments. User productivity will increase and frustration will decrease by having a single user environment to learn and use.

For example, Windows applications use the Win32 API, which is implemented by the Win32 subsystem. Programs written for MS-DOS, OS/2, Microsoft Windows version 3.x, and POSIX run in their own environmental subsystems, all of which interact extensively with the Win32 subsystem to implement their functionality.

The Interix subsystem implements POSIX APIs. Even with this independence, you can still run Win32 programs, such as Notepad (Notepad.exe) and Calculator (Calc.exe), from the Interix shell prompt.

patterns & practices
proven practices for predictable results
